Available online at www.sciencedirect.com

ScienceDirect

Procedia Engineering 61 (2013) 57 – 62

Procedia Engineering

www.elsevier.com/locate/procedia

Parallel Computational Fluid Dynamics Conference (ParCFD2013)

Solving Two-Dimensional Euler Equations on GPU

Xiaofeng He[a], Zheng Wang[a], Tiegang Liu[a,*]

[a]*LMIB and School of Mathematics and Systems Science, Beihang University, Beijing 100191, P. R. China.*

Abstract

The potential of GPU in scientific computation has been tested in many applications during recent years. CUDA, short for Compute Unified Device Architecture, is a solution of GPGPU proposed by Nvidia Corporation. Researchers can improve performance of programs and reduce time cost with CUDA involved.

This paper presents a flow solver for two-dimensional airfoil governed by Euler equations on GPU using CUDA. Body-fitted structured mesh is employed here, and finite-volume method is used to obtain the solution. Since the number of concurrent threads supported by CUDA is very huge(more than 10^{15}), the proposed solver uses a CUDA thread to calculate the flow variables for each cell of the mesh. Two dimensional CUDA blocks are used to manage threads in the application. This paper gives a performance comparison between GPU parallel implementation and CPU serial implementation, and the results show that the proposed method has good efficiency. As CUDA supports three dimensional thread blocks, the solver can be extended to three dimensional cases naturally.

Selection and peer-review under responsibility of the Hunan University and National Supercomputing Center in Changsha (NSCC)

Keywords: CUDA, Euler equations, GPU

1. Introduction

Computational fluid dynamics(CFD) plays a more and more important role in the development of new products in the aeronautical industry[1]. With the progresses in the last decades, CFD can simulate the realistic scenarios of many cases. But to complex systems, algorithms with high accuracy are required to handle them. Despite the increasing performance of computers, it often takes long time to solve a complicated system of fluid dynamics, which slows down the progress of CFD development.

Many technologies are proposed to reduce the time cost by CFD flow solvers[2]. Parallel computing is a good idea to accelerate programs, as it reduces the burden of per processor by increasing the number of processor. MPI is a mature technology in industry[3], but it's costly to set up a high performance MPI cluster and thus it's expensive to use frequently.

Compared with MPI, a better choice is GPGPU[4]. GPGPU has seen a tremendous increase in applications over the past few years[5][6][7]. CUDA, short for Compute Unified Device Architecture, is a solution of GPGPU proposed by Nvidia Corporation. There are several advantages to prefering CUDA to MPI. Firstly, it's cheap to use CUDA. One can buy a GPU with several hundred dollars now, which consists of hundreds or even more than a thousand cores.

* Corresponding author. Tel.: +86-135-2225-7421.
E-mail address: liutg@buaa.edu.cn

Selection and peer-review under responsibility of the Hunan University and National Supercomputing Center in Changsha (NSCC)
doi:10.1016/j.proeng.2013.07.093

One can deploy the CUDA environment with laptop, as GPU is part of the high performance video cards in general. Secondly, performance of CUDA is very good. With the development of GPU technologies, CUDA now is widely used. Thirdly, it's easy to code and debug. CUDA toolkits work well with many visualization IDE such as Microsoft visual Studio or Eclipse. There are also lots of other arguments for CUDA, such as being kind to environment, etc.

In this paper, solving two-dimensional Euler equations with CUDA is described. The remainder of the chapter is organized as follows: Euler equations and a flow solver will be introduced first. After that, a model on how to implement the flow solver on GPU is built. Results of CUDA program will be given to compare with CPU's.

2. Euler equations and solver

We will consider the Euler equations in this paper:

$$\frac{\partial}{\partial t}\int_{\Omega} W d\Omega + \oint_{\partial\Omega} F_c dS = 0$$

with

$$W = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{bmatrix}, \qquad F_c = \begin{bmatrix} \rho V \\ \rho u V + n_x p \\ \rho v V + n_y p \\ \rho H V \end{bmatrix},$$

where

$$E = \frac{p}{\rho(\gamma - 1)} + \frac{1}{2}V^2, \quad H = E + \frac{p}{\rho}, \quad V = u n_x + v n_y,$$

and $\vec{n} = (n_x, n_y)$ denotes the unit normal vector at each cell boundary, W is conservative variables, F_c is a vector of convective fluxes, V is the contravariant velocity, which is normal to the surface element dS, H is the total enthalpy.

The well known JST(Jameson-Schmidt-Turkel) finite volume scheme[8] is employed in this work. The conservation system is semi-discretized and a second order spatial central discretization is applied to the flux, explicit four-stage Runge-Kutta method is used to solve the semi-discretized equations until the steady state is obtained.

3. CUDA programming model

3.1. Overview

For a system consists of CPU and GPU, CPU is named "host" and GPU is called "device"[9]. CPU is the owner of whole computer system, including GPU, and GPU is just a part of computer system. Both CPU and GPU has its own private memories and other resources. Function executed on host is called "host function". A function called by CPU while executed on GPU is named "kernel function". As GPUs locate outside of CPU in general, kernel functions are asynchronous to host by default. In most cases, this feature is very useful: CPU sends instructions to GPUs for execution and continues running codes irrelevant. It reduces the total time consumption via overlapping execution of CPU and GPU codes. Sometimes one may need to synchronize between CPU and GPU, and CUDA implements this by blocking the execution of CPU instructions until GPU has finished certain requests.

A function executed on GPU(namely kernel function) has the following format(two dimensional case):

```
__global__ void kernelFun(typname args)
{
        // thread indexes
        int tid = threadIdx.x + blockIdx.x*blockDim.x;
        int tjd = threadIdx.y + blockIdx.y*blockDim.y;

        // some meaningful codes and do something
}
```

The key word "__global__" declares that this function is called by CPU while executed on GPU, and threadIdx, blockDim are built-in structs in CUDA. With GPU involves, data transfer between CPU and GPU is required. So the flow chart of a GPU program looks like this:

```
int main(int argc, char* argv[])
{
        // variables declaration

        // initialize data

        // transfer data to GPU

        kernelFun<<<blocksize, threadsize, memsize, streamid>>>(args);

        // transfer data to CPU and synchronize

        // repeat until satisfy certain conditions

        // output solution

       return 0;
}
```

General speaking, A GPU has much more cores than a CPU. For example, Nvidia Tesla M2070 has 448 CUDA cores, while Intel Xeon E5620 has only 4 cores. It's wise to leave calculation burdens to GPU and this should be done for global performance consideration of an application.

3.2. CUDA threads and mesh cells

Just like CPU, a thread in CUDA is a branch of the process. CUDA supports concurrent execution of a large amount of threads and via this way it accelerates the calculation. The "container" of threads is called "block", and a grid is consists of a set of blocks. Due to hardware restrictions, the number of threads in a block and the number of blocks in a grid are limited. For a GPU with capability 2.x, maximum number of threads in a kernel function is larger than 10^{15}(the actual number is $1024 * 60356^3$). It's a impressive and inspiring fact.

Both block and grid are three-dimensional, this feature is perfect for solving three-dimensional problems. It's of course capable of dealing with one-dimensional and two-dimensional problems too. If a thread has index (x, y, z), then its global index *tid* within a grid is

$$tid = x + y \times blockDim.x + z \times blockDim.y \times blockDim.x.$$

For two dimensional CUDA blocks, z is always 0, then the global thread index becomes

$$tid = x + y \times blockDim.x.$$

Notice that the thread in CUDA is indexed along x-axis first, y-axis second, and z-axis in the last.

Mesh employed in this paper is two-dimensional body-fitted structured mesh. If a cell has index (i, j), then its global index

$$gid = i + j \times icells\ .$$

In this equation, *icells* denotes the number of cells along the i direction. Flow variables associated with each cell are required to be obtained. For a serial CPU program, it has to run at least $i \times j$ times to retrieve and update data among the whole computation domain for every time step. If i,j is big, the loads is costly.

As a thread block in CUDA can be made up of two-dimensional threads, the mesh can be divided into several portions and each portion is a two dimensional block. Due to the huge amount of thread CUDA supports, it's reasonable

to relate a thread to a cell for performance consideration. Thus map relation between cells and threads is built as Fig. 1.

In this work, a thread is assigned to calculate flow variables associated with an element. The complexity of algorithm on each core is decreased and performance of the application is improved by using *ICELLS*× *JCELLS* CUDA threads, where ICELLS and JCELL means the number of cells along the *i* and *j* direction. Another attention is that how to index the cells within mesh. According to the way of CUDA threads indexing and for memory alignment consideration, cells should be indexed first along x-axis and then y-axis[11]. Thus CUDA programming model is built and can be used to solve the flow.

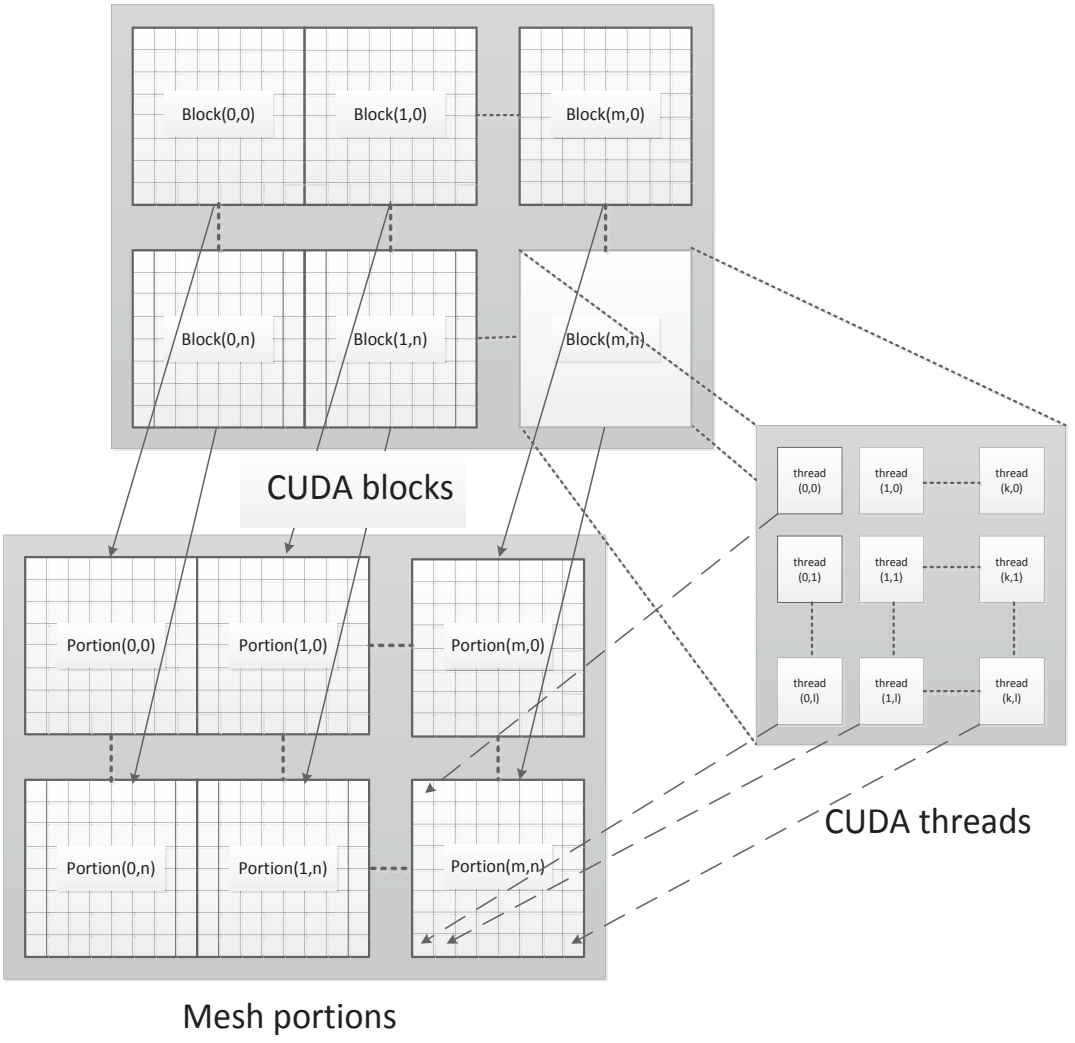

Fig. 1. mapping of CUDA threads and mesh cells.

4. Results

A NACA0012 wing in transonic($M_\infty = 0.8, \alpha = 1.25°$) flow is used as a test case. Body-fitted structured mesh is applied in the both CPU and GPU solvers, which is shown in Fig. 2.

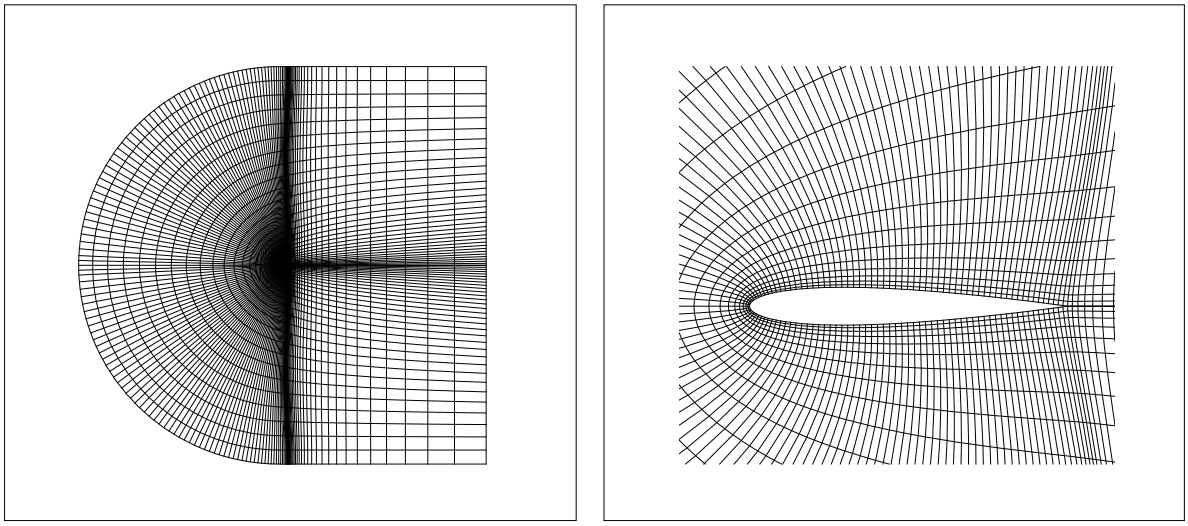

Fig. 2. body-fitted structured mesh.

Both CPU and GPU codes are executed on same machine with same loads. Specification of CPU and GPU are shown in following table:

Table 1. specification of CPU and GPU.

Item	CPU	GPU
Model	Intel Xeon E5620	Tesla M2070
Clock rate(Hz)	2.4G	1.15G
Capability	-	2.0
Cores	4	14×32=448
Memory	8GB	6GB

This table implies advantages of CPU and GPU: the clock rate of CPU is much higher and owes more random access memory(RAM) in general cases, while GPU has much more cores. If one can make good use of GPU's potentia, GPU will release high performance. Time expense of CPU's serial program GPU's program are shown in Table. 2:

Table 2. Results of CPU and GPU solver.

Grid size	CPU Time(s)	GPU Time(s)	Speedup
448×160	219.94	22.96	9.6

According to CUDA features, the acceleration is better when scale of problem increasing. In other words CUDA will be more effective with more complicated calculation involved.

The pressure coefficient of flow field obtained from CPU and GPU programs are shown in Fig. 3.

One may notice that around the corners of curves there are small differences as shown in Fig. 3. Reason for those differences is that accuracy of GPU is not so high compared with CPU, but relative errors are IEEE-compliant[10].

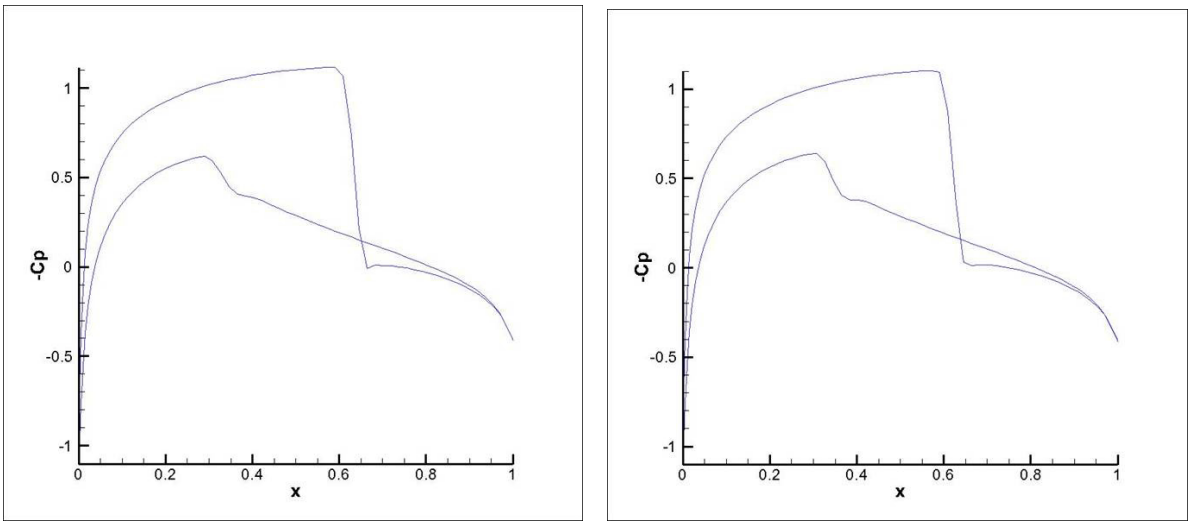

Fig. 3. Pressure coefficient from (a) CPU and (b) GPU.

5. Conclusion and Future work

In this paper, a flow solver based on CUDA is developed for two-dimensional airfoil governed by Euler equations. Body-fitted structured mesh is employed here, and finite-volume method is used to obtain the solution. CUDA accelerates the program and gives a satisfactory result. Since the architecture of CUDA aims at three-dimensional simulation, the solver can be extended to solve three-dimensional problems naturally.

The development of complicated three-dimensional flow solvers are ultimate aim of CFD[12], CUDA will play an more important role in designing fast CFD solver in the future. New solvers based on high-accuracy algorithms such as discontinuous Galerkin method[13] are under development.

References

[1] A. Jameson, Kui Ou, 50 years of transonic aircraft design, Progress in Aerospace Sciences, Volume 47, Issue 5, p. 308-318, 2011.
[2] Markus Nordén, Sverker Holmgren, Michael Thuné, OpenMp versus MPI for PDE solvers based on regular sparse numerical operators, Future Generation Computer Systems, Volume 22, Issues 1-2, p 194-203, 2006.
[3] Paul T. Lin, John N. Shadid, Towards large-scale multi-socket, multicore parallel simulations: Performance of and MPI-only semiconductor device simulator, Journal of Computational Physics, Volume 229, p 6804-6818, 2010.
[4] David B. Kirk, Wen-mei W. Hwu, 2010. Programig massively parallel processors, Tsinghua University Press, Beijing, China.
[5] Dana A.Jacobsen, Julien C. Thibault, Inanc Senocak, An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters, 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, 2010.
[6] Christopher P. Stone, Earl P.N Duque, Yao Zhang, David Car, John D. Owens, Roger L. Davis, GPGPU parallel algorithms for structured-grid CFD codes, 20th AIAA Computational Fluid Dynamics Conference, 2011.
[7] Wei Ran, Wan Cheng, Fenghua Qin, Xisheng Luo, GPU accelerated CESE method for 1D shock tube problems, Journal of Computational Physics, Volume 230, Issue 24, p 8797-8812, 2011.
[8] A. Jameson, W. Schmidt, and E. Turkel, Numerical solutions of the Euler equations by Finite volume methods using Runge-Kutta time-stepping schemes, AIAA Paper 81-1259, 1981.
[9] Jason Sanders, Edward Kandrot, 2010. CUDA By Example: an Introduction to General-Purpose GPU Programming, Addison-Wesley Educational Publishers Inc, Boston, USA.
[10] NVIDIA: NVIDIA CUDA C Programming Guide, NVIDIA Corporation, May, 2011.
[11] Neil G. Dickson, Kamran Karimi, Firas Hamze, Importance of explicit vectorization for CPU and GPU software performance, Journal of Computational Physics, Volume 230, Issue 13, P 5383-5398, 2011.
[12] Ali Khajeh-Saeed, J. Blair Perot, Direct numerical simulation of turbulence using GPU accelerated supercomputers, Journal of Computational Physics, Volume 235, p 241-257, 2013.
[13] Bernardo Cockburn, Chi-Wang Shu, The Runge-Kutta Discontinuous Galerkin Method for Conservation Laws V: multidimensional Systems, Journal of Computational Physics, Volume 141, Issue 2, p 199-224, 1998.